Tema 8: Construcción de tipos de datos

Índice

1. Introducción

Los beneficios de introducir tipos de datos en un lenguaje de programación son muy grandes. Entre ellos se encuentran la posibilidad de detectar y prevenir errores en las llamadas a funciones o en la asignación de valores a variables. En este tema veremos algunas estrategias para introducir tipos de datos en Scheme.

Hemos visto en el tema 4 las ventajas de usar la abstracción de datos, definiendo un nombre para una composición de datos primitivos. Por ejemplo, hemos hecho esto con los árboles o con las cartas. Hemos creado una estructura de datos utilizando parejas (secuencia de *cons*) y hemos definido unos constructores y unos selectores. Los constructores son funciones que devuelven un dato compuesto (un árbol o una carta) creado a partir de unos valores dados (que pueden ser también compuestos) y los selectores devuelven los datos elementales que forman el dato compuesto.

Por ejemplo, el constructor

```
(make-arbol dato hijos)
```

construye un árbol a partir de un dato y una lista de árboles hijos (que puede ser vacía, para el caso de los nodos hoja). A su vez, los selectores

```
(dato arbol)
(hijos arbol)
```

devuelven el dato asociado a un árbol y su lista de hijos.

De esta forma, definiendo constructores y selectores, podemos abstraer el tipo de datos y usarlo sin preocuparnos de la implementación.

Pero esta solución no es perfecta. Su limitación fundamental es que la abstracción que definimos sólo existe en la disciplina del programador, no hay nada en el lenguaje que la refuerce ni la represente. Para Scheme el dato construido sólo es un conjunto de parejas a las que se podría acceder utilizando las funciones para parejas `car` y `cdr`. Además, tampoco es posible comprobar si un dato es de un tipo determinado, lo que nos puede llevar a errores como el de llamar a una función usando argumentos que no son del tipo correcto.

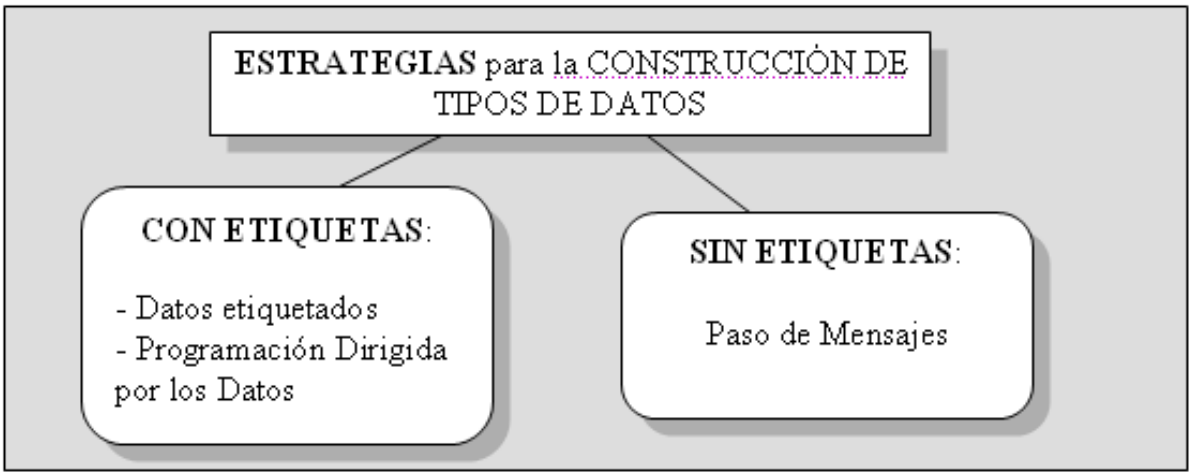
El problema de esta solución se debe en parte a que Scheme es un lenguaje *débilmente tipeado* (*weakly typed language* en inglés). En Scheme las variables no tienen un tipo asociado. Una variable puede ligarse a un dato de un tipo y después a un dato de otro tipo. Para especificar una función no es necesario definir los tipos de los parámetros ni del valor devuelto. En un lenguaje déblimente tipeado no se comprueban los tipos de las variables en tiempo de compilación o ni siquiera existen. Otros lenguajes débilmente tipeados son **Smalltalk** o **Python**.

En el otro extremo se encuentran lenguajes como **Pascal** o **Ada**, en los que todas las variables y parámetros tienen asociados un tipo de datos y el compilador comprueba que todas las asignaciones y paso de parámetros son consistentes. A estos lenguajes se les denomina *fuertemente tipeados* (*strongly typed languages* en inglés).

Vamos a buscar una solución dinámica a los tipos en Scheme: vamos a hacer que los datos (no las variables) estén tipeados (contenga información de su tipo). O sea, dado un dato podremos consultar de qué tipo de dato es. Esto permitira:

- chequear que los argumentos se corresponden con los tipos esperados
- definir procedimentos genéricos o polimórficos (que reciben más de un tipo)

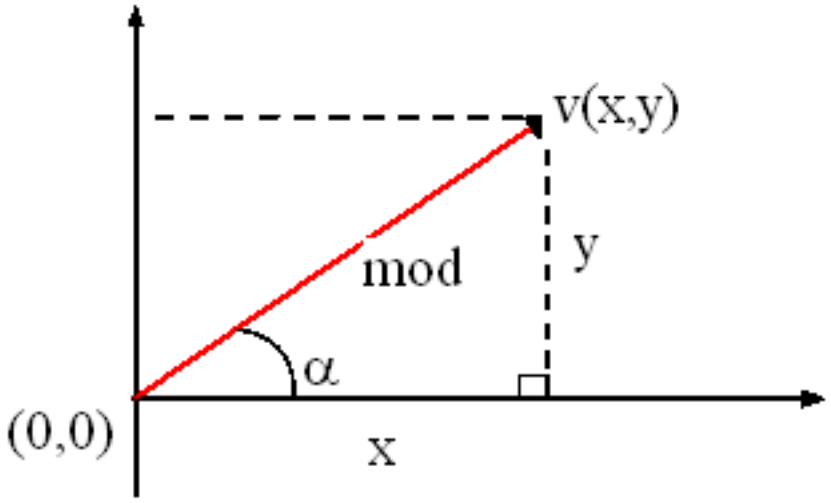
Vamos a hacer esto utilizando principalmente dos estrategias. En la primera, la más sencilla, etiquetaremos los datos con un identificador que definirá su tipo. En la segunda implementaremos los datos como funciones que aceptan mensajes. Será un primer paso para la programación orientada a objetos. La primera estrategia se establece poniendo etiquetas a los datos y la podemos dividir en dos: *datos etiquetados* y *programación dirigida por los datos*. A la segunda estrategia la denominamos *paso de mensajes*.



2. Vectores

Vamos a utilizar como ejemplo la representación de vectores. Se pueden representar de dos formas: utilizando el módulo y el ángulo del vector (notación polar) o utilizando sus coordenadas (notación cartesiana). Consideramos que el origen de los vectores es el (0,0).

En la siguiente figura se muestran las dos representaciones de un vector.



Es sencillo pasar un vector de una representación a otra. De cartesiana a polar:

```
mod = sqrt (x^2 + y^2)
alfa = atan(y,x)
```

De polar a cartesiana:

```
x = mod * cos(alfa)
y = mod * sin(alfa)
```

Definimos las siguientes funciones de Scheme que realizan esta transformación:

```
(define (xy->modulo coor-x coor-y)
   (sqrt (+ (square oor-x) (square coor-y))))

(define (xy->angulo coor-x coor-y)
   (atan coor-y coor-x))

(define (mod-ang->coor-x mod ang)
   (* mod (cos ang)))

(define (mod-ang->coor-y mod ang)
   (* mod (sin ang)))
```

3. Operaciones con vectores

Vamos a considerar la suma y la multiplicación de vectores. La suma de dos vectores es muy

sencilla utilizando a representación cartesiana, mientras que para la multiplicación es conveniente utilizar la representación polar.

Dados dos vectores `v1=(a1,b1)` y `v2=(a2,b2)` utilizando notación cartesiana, su **suma** se calcula como:

```
v1+v2 = (a1+a2, b1+b2)
```

Dados dos vectores `v1=(r1,alfa1)` y `v2=(r2,alfa2)` representados en notación polar, su **multiplicación** se calcula como:

```
v1*v2 = (r1*r2, alfa1+alfa2)
```

4. Programando con vectores

Hemos visto que es posible representar los vectores con dos notaciones (polar y cartesiana) y que dependiendo de qué operaciones se realizan con los vectores es conveniente usar una representación u otra.

Si queremos escribir un programa en Scheme que utilice vectores podríamos pensar que tenemos que decidirnos por una sola notación. Pero en programación lo importante es poder tomar este tipo de decisión en cualquier momento y poder retrasar este tipo de decisiones el máximo tiempo posible: cuando tengamos el programa terminado y las pruebas nos indican que una notación es preferible a otra (por cuestiones de eficiencia, por ejemplo).

Para poder usar ambos tipos de datos simultáneamente:

1. Definimos ambas representaciones mediante dos tipos de datos: `cart` y `polar`. Ambos tipos de datos tienen una barrera de abstracción común, aunque especializada en datos del tipo correspondiente.

   Así, definimos para el tipo de dato `cart` las siguientes funciones:

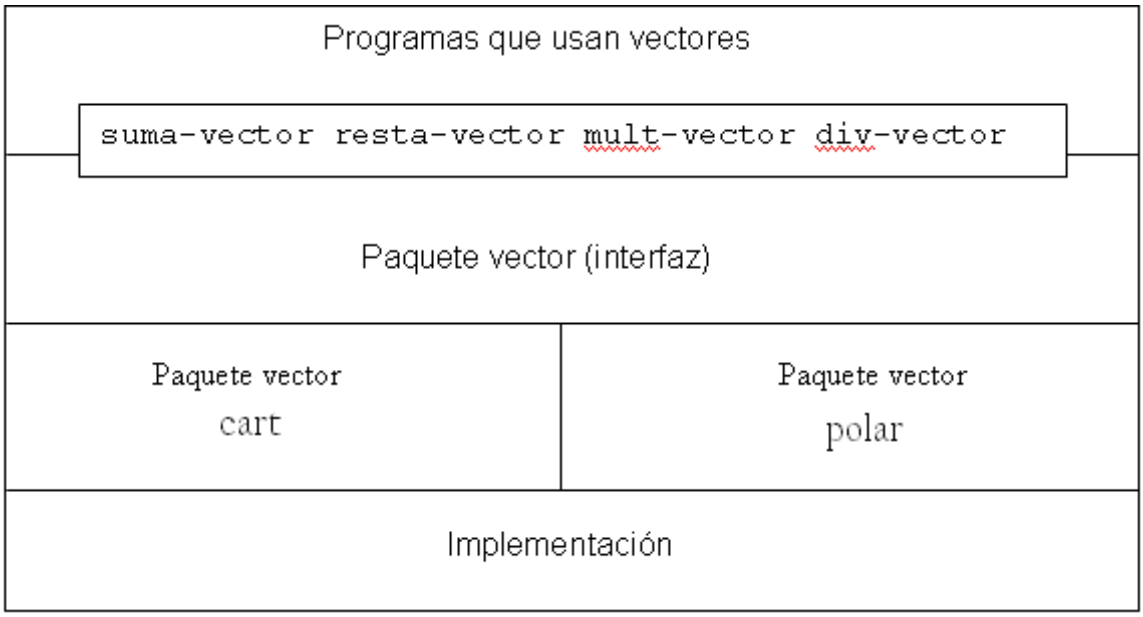
   - (make-from-x-y-cart x y)
   - (make-from-mod-ang-cart mod ang)
   - (x-cart v-cart)
   - (y-cart v-cart)
   - (mod-cart v-cart)
   - (ang-cart v-cart)

   Y para el tipo de dato `polar` definimos las funciones:

   - (make-from-x-y-polar x y)
   - (make-from-mod-ang-polar mod ang)
   - (x-polar v-polar)

- (y-polar v-polar)
- (mod-polar v-polar)
- (ang-polar v-polar)

2. Definimos una barrera de abstracción genérica que permita utilizar ambos tipos de datos de forma indistinta:
   - (make-from-x-y-vector x y)
   - (make-from-mod-ang-vector mod ang)
   - (x-vector v)
   - (y-vector v)
   - (mod-vector v)
   - (ang-vector v)
3. Por último, definimos una barrera de abstracción de más alto nivel que define operaciones sobre vectores utilizando la barrera de abstracción anterior:
   - (suma-vector v1 v2)
   - (resta-vector v1 v2)
   - (mult-vector v1 v2)
   - (div-vector v1 v2)



Las funciones genéricas de alto nivel se implementarían como (comunes a todas las estrategias):

```
(define (suma-vector v1 v2)
  (make-from-x-y-vector (+ (x-vector v1) (x-vector v2))
                        (+ (y-vector v1) (y-vector v2))))

(define (resta-vector v1 v2)
  (make-from-x-y-vector (- (x-vector v1) (x-vector v2))
                        (- (y-vector v1) (y-vector v2))))

(define (mult-vector v1 v2)
  (make-from-mod-ang-vector (* (mod-vector v1) (mod-vector v2))
                            (+ (ang-vector v1) (ang-vector v2))))

(define (div-vector v1 v2)
  (make-from-mod-ang-vector (/ (mod-vector v1) (mod-vector v2))
                            (- (ang-vector v1) (ang-vector v2))))
```

Los constructores genéricos de vectores para ambas representaciones serían (comunes a todas las estrategias):

```
(define (make-from-x-y-vector x y)
  (make-from-x-y-cart x y))

(define (make-from-mod-ang-vector mod ang)
  (make-from-mod-ang-polar mod ang))
```

Vamos a ver a continuación cómo implementar los paquetes vector mediante las distintas estrategias (sin tener que modificar las funciones genéricas).

5. Tipos de datos etiquetados

En este apartado explicaremos cómo representar tipos de datos utilizando datos etiquetados. Vamos a utilizar la representación de vectores como ejemplo.

5.1. Representación de tipos datos con etiquetas

La primera estrategia que vamos a definir para representar tipos de datos consiste en añadirle a todos los datos una *etiqueta* con un identificador del tipo de dato. Para implementarlo en Scheme construimos una pareja (*cons*) cuya parte izquierda es el identificador del tipo de dato y cuya parte derecha es el dato propiamente dicho.

Para implementar esta estrategia definimos tres funciones en Scheme. La función *(attach type-tag contents)* recibe como parámetros una etiqueta de tipo y un dato y devuelve un dato etiquteado. La función *(type-tag datum)* toma como parámetro un dato etiquetado y devuelve su tipo. Por último, la función *(contents datum)* toma como parámetro un dato etiquetado y devuelve el dato propiamente dicho.

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "error en type-tag" datum " no es un dato etiquetado")))

(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "error en contents" datum " no es un dato etiqueteado")))
```

Un ejemplo de utilización es el siguiente, aplicado al ejemplo de las cartas de la baraja del tema 4:

```
(define c1 (attach-tag 'carta (make-carta 5 'oros)))
(type-tag c1) -> 'carta
(contents c1) -> la carta propiamente dicha
```

Para poder definir las funciones genéricas es necesario reconocer el tipo de dato que llega como parámetro. En la función genérica se recibirá un dato de tipo `cart` o `polar` y se deberá llamar a la función específica correspondiente. Para ello usamos la estrategia de tipos etiquetados, y en los constructores añadimos una etiqueta al dato indicando de que tipo es. Todas las funciones trabajan con datos etiquetados.

La implementación de los vectores con la notación cartesiana es la siguiente.

```
(define (make-from-x-y-cart x y)
  (attach-tag 'cart (cons x y)))

(define (make-from-mod-ang-cart mod ang)
  (attach-tag 'cart
              (cons (mod-ang->coor-x mod ang)
                    (mod-ang->coor-y mod ang))))

(define (x-cart v) (car v))

(define (y-cart v) (cdr v))

(define (mod-cart v)
  (xy->modulo (x-cart v)
              (y-cart v)))

(define (ang-cart v)
  (xy->angulo (x-cart v)
              (y-cart v)))
```

La implementación de los vectores con la notación polar es la siguiente.

```
(define (make-from-x-y-polar x y)
  (attach-tag 'polar
              (cons (xy->modulo x y)
                    (xy->angulo x y))))

(define (make-from-mod-ang-polar mod ang)
  (attach-tag 'polar (cons mod ang)))

(define (mod-polar v) (car v))

(define (ang-polar v) (cdr v))

(define (x-polar v)
  (mod-ang->coor-x (mod-polar v) (ang-polar v)))

(define (y-polar v)
  (mod-ang->coor-y (mod-polar v) (ang-polar v)))
```

Las funciones genéricas de nivel más bajo se implementan como sigue.

```
(define (cart? v)
  (eq? (type-tag v) 'cart))

(define (polar? v)
  (eq? (type-tag v) 'polar))

(define (x-vector v)
  (cond ((cart? v)
         (x-cart (contents v)))
        ((polar? v)
         (x-polar (contents v)))
        (else (error "Tipo desconocido: -- COOR-X" v))))

(define (y-vector v)
  (cond ((cart? v)
         (y-cart (contents v)))
        ((polar? v)
         (y-polar (contents v)))
        (else (error "Tipo desconocido: -- COOR-Y" v))))

(define (mod-vector v)
  (cond ((cart? v)
         (mod-cart (contents v)))
        ((polar? v)
         (mod-polar (contents v)))
        (else (error "Tipo desconocido: -- MODULO" v))))

(define (ang-vector v)
  (cond ((cart? v)
         (ang-cart (contents v)))
        ((polar? v)
         (ang-polar (contents v)))
```

```
        (else (error "Tipo desconocido: -- ANGULO" v))))
```

6. Programación dirigida por los datos

Necesitamos una tabla *hash* (estructura de datos que asocia claves con valores) global indexada por dos claves para guardar en ella las funciones específicas. Esta tabla se implementa en Scheme con las siguientes funciones. No nos interesa la implementación, sino los ejemplos finales en los que se muestra cómo usar las funciones `put` y `get` que guardan y recuperan datos en la tabla global.

```
(define nil '())

(define (get-primitive key)
  (let ((record (assq key (cdr the-table))))
    (if (not record)
    nil
    (cdr record))))

(define (put-primitive key value)
  (let ((record (assq key (cdr the-table))))
    (if (not record)
    (set-cdr! the-table
          (cons (cons key value)
            (cdr the-table)))
    (set-cdr! record value)))
  'ok)

(define the-table (list '*table*))

(define (search-value key list)
   (cond
      ((null? list) #f)
      ((equal? key (car (car list))) (cdr (car list)))
      (else (search-value key (cdr list)))))

(define (put key1 key2 value)
   (let ((l-value (get-primitive key1)))
      (if (pair? l-value)
         (put-primitive key1 (cons (cons key2 value) l-value))
         (put-primitive key1 (list (cons key2 value))))))

(define (get key1 key2)
   (let ((l-value (get-primitive key1)))
      (search-value key2 l-value)))


; Ejemplos de uso de estas funciones

(put 'a 'a 10)
(put 'a 'b 20)
(put 'b 'b 30)
```

```
(get 'a 'a)
(get 'a 'b)
(get 'b 'b)
(put 'b 'b (get 'a 'a))
(get 'b 'b)
```

Usamos la tabla global para guardar las funciones específicas de los distintos tipos de datos. Asociamos cada función específica a la función genérica que implementa y al tipo de dato del argumento de la función.

```
(put 'x 'cart x-cart)
(put 'y 'cart y-cart)
(put 'mod 'cart mod-cart)
(put 'ang 'cart ang-cart)
(put 'make-from-x-y 'cart make-from-x-y-cart)
(put 'make-from-mod-ang 'cart make-from-mod-ang-cart)

(put 'x 'polar x-polar)
(put 'y 'polar y-polar)
(put 'mod 'polar mod-polar)
(put 'ang 'polar ang-polar)
(put 'make-from-x-y 'polar make-from-x-y-polar)
(put 'make-from-mod-ang 'polar make-from-mod-ang-polar)
```

	Tipos		
Operaciones		**cart**	**polar**
	x	x-cart	x-polar
	y	y-cart	y-polar
	mod	mod-cart	mod-polar
	ang	ang-cart	ang-polar

Tabla de operaciones para el sistema de vectores

Teniendo esta tabla es muy sencillo definir una función `operate` que tiene como argumento un nombre de función genérica y un tipo de dato y devuelve la función específica a aplicar a ese tipo de dato para implementar la función genérica.

```
(define (operate op obj)
```

```
  (let ((proc (get op (type-tag obj))))
    (if proc
       (proc (contents obj))
       (error "Unknown operator for type"))))
```

Una vez implementada la función `operate` hay que reescribir las funciones genéricas como llamadas a esta función. De esta forma la barrera de abstracción de los datos continua siendo la misma.

```
(define (x-vector v) (operate 'x v))
(define (y-vector v) (operate 'y v))
(define (mod-vector v) (operate 'mod v))
(define (ang-vector v) (operate 'ang v))
(define (type v) (operate 'type v))
```

Por último, para los constructores accedemos directamente a la tabla:

```
(define (make-from-mod-ang-vector mod ang)
   ((get 'make-from-mod-ang 'polar) mod ang))

(define (make-from-x-y-vector x y)
   ((get 'make-from-x-y 'cart) x y))
```

7. Representación de tipos datos sin etiquetas: Paso de mensajes

La estrategia del paso de mensajes para codificar los tipos de datos es la segunda que vamos a ver.

```
;; Paquete de representacion con coordenadas

(define (make-from-x-y-vector x y)
  (lambda (op)
    (cond ((eq? op 'x) x)
          ((eq? op 'y) y)
          ((eq? op 'mod)
           (xy->modulo x y))
          ((eq? op 'ang)
           (xy->angulo x y))
          ((eq? op 'type) 'cart)
          (else
           (error "Unknown op -- MAKE-FROM-COOR-X-COOR-Y" op)))))

;; Paquete de representación modulo-angulo
(define (make-from-mod-ang-vector  mod ang)
  (lambda (op)
    (cond ((eq? op 'x)
           (mod-ang->coor-x mod ang))
          ((eq? op 'y)
           (mod-ang->coor-y mod ang))
```

```
            ((eq? op 'mod) mod)
            ((eq? op 'ang) ang)
            ((eq? op 'type) 'polar)
            (else
             (error "Unknown op -- MAKE-FROM-MOD-ANG" op)))))

  ;; Funciones genéricas que funcionan con cualquier tipo de datos

  (define (x-vector v) (v 'x))
  (define (y-vector v) (v 'y))
  (define (mod-vector v) (v 'mod))
  (define (ang-vector v) (v 'ang))
  (define (type v) (v 'type))

  ;
  ; código que prueba el módulo
  ;

  (define v1 (make-from-x-y-vector 5 10))
  (mod-vector v1)
  (ang-vector v1)
  (define v2 (make-from-mod-ang-vector (mod-vector v1) (ang-vector v1)))
  (x-vector v2)
  (y-vector v2)
  (define v3 (suma-vector v1 v2))
  (x-vector v3)
  (y-vector v3)
  (mod-vector v3)
  (ang-vector v3)
  (define v4 (mult-vector v1 v2))
  (x-vector v4)
  (y-vector v4)
  (mod-vector v4)
  (ang-vector v4)
```

8. Estructuras en MzScheme

MzScheme es la versión de Scheme que se define en DrScheme. En esta versión se definen extensiones del lenguaje básico como son las estructuras, la programación orientada a objetos o las comunicaciones de red

En MzScheme se pueden construir estructuras con campos. Definimos la estructura `vector-cart` y se definen automáticamente el constructor y las funciones de acceso a los campos `x` e `y`.

```
  (define-struct vector-cart (x y))

  (define v1 (make-vector-cart 5 3))
```

```
(vector-cart-x v1) -> 5
(vector-cart-y v1) -> 3
```

Ahora definimos las funciones que obtienen el módulo y el ángulo del vector: `vector-cart-mod` y `vector-cart-ang` que usan las funciones auxiliares definidas previamente:

```
(define (vector-cart-mod v)
   (xy->modulo (vector-cart-x v)
               (vector-cart-y v)))
(define (vector-cart-ang v)
   (xy->angulo (vector-cart-x v) (vector-cart-y v)))

(vector-cart-mod v1)
(vector-cart-ang v1)
```

Definimos la estructura `vector-polar` con los campos `mod` y `ang` que representa el vector en forma polar. Definimos también las funciones `vector-polar-x` y `vector-polar-y` que obtienen las coordenadas x e y del vector usando las funciones auxiliares definidas al principio del tema.

```
(define-struct vector-polar (mod ang))

(define (vector-polar-x v)
   (mod-ang->coor-x (vector-polar-mod v) (vector-polar-ang v)))

(define (vector-polar-y v)
   (mod-ang->coor-y (vector-polar-mod v) (vector-polar-ang v)))

; ejemplos de uso

(define v2 (make-vector-polar 5.830951894845301 0.5404195002705842))
(vector-polar-x v2)
(vector-polar-y v2)
```

9. Referencias

Para saber más de los temas que hemos tratado en esta clase puedes consultar las siguientes referencias:

- Structure and Interpretation of Computer Programs (http://mitpress.mit.edu/sicp/full-text/book/book.html) , Abelson y Sussman, MIT Press 1996 (pp. 169-188). Disponible biblioteca politécnica (acceso al catálogo (http://gaudi.ua.es/uhtbin/boletin/285815))